INTRODUCTION

After the implementation of the Konnichiwa, we had to make some changes from the architectural design that we made before. What and why will be described in this document, along with this a tutorial on how to install and use the program. Our main focus throughout this project was testability, we chose pattern for this reason and we will present our test results along with comments and problems through the development.

DESIGN DETAILS/IMPLEMENTATION DETAILS

Since we decided on using the MVC pattern, we have split the architecture in three parts, model, view and controller. The decision on using the MVC pattern is the result from the requested quality attribute testability, as dividing functionality in layers makes it easier to test each part. In Android, the view and controller parts are more or less intertwined – the XML files in res/layout describes the graphical interface, but its content is decided in the controller, which can also be used to affect the GUI.

Our main controller class is Konnichiwa, which is in charge of starting the application, allowing the user to log in, and then utilizes the NetworkHandler class to populate the model ContactList with the current contact list (ContactList is an ArrayList of Contact elements), which is fetched from Google's servers. It then starts the UserList activity if the user is successful in logging in. The Konnichiwa class also has functionality for testing the Internet connection.

When the UserList activity is opened, it gets the contact list from the model, and displays it in a ListView, getting information from each Contact, via the ContactList, about whether or not each contact is available. When one of these list elements, or contacts, is selected, a Chat activity opens.

The Chat activity gets the previous messages from the database, and adds itself as a listener to the Conversation class, which is a singleton that has values that get reset every time you close a chat window. When the NetworkHandler receives new messages, it adds them to the conversation model, which again fires a PropertyChangeEvent, so that Chat updates its GUI. When the user chooses to go back to the UserList, the UserList is generated….

The architectural decisions we made, lets us add new functionalities. It should be possible to add other functions to our app like connect to other chat servers, send files or view a picture of the contact you are having a conversation with, these and more functions possible to add to the application without much hassle. But because of limited time we implemented a robust basic chat client.

USER'S MANUAL

INSTALLING THE PROGRAM (IN ECLIPSE)

1. Install the Android SDK
2. Import the Konnichiwa chat client
3. Import aSmack into library.
4. Left click the project folder and press run as android application
5. Wait for it to start (ca 5 minutes)

USING THE PROGRAM

1. Log in to your Gmail account or use one of our test accounts.
2. Select between viewing your online or offline friends
3. Select your online friend
4. Start a wonderful chat with your friend

   Note: Messages are forwarded to the contact as you input them, but will be viewed in the chat client with status, Incoming and Receiving

   Test accounts: progarktest@gmail.com, progarktest1@gmail.com, progarktest2@gmail.com, progarktest3@gmail.com, progarktest4@gmail.com password: 81549300

TEST REPORT

UNIT TESTING

Tests the robustness of a method in a class.

Class: Example class

Method name	**Description**	**Input**	**Results**	**Changes**
exampleMethod()	Example	NULL	NullpointException	Added try/catch
exampleMethod()	Example	NULL	Approved!	

INTEGRATION TESTING

Class A uses Method A to call Method B in Class B.

Class A	Class B	Method A	Method B	Input	Results	Changes
ExClassA	ExClassB	sMethod()	rMethod()	Unexpected arraysize	ArrayOutOfBounds	Added array check
ExClassA	ExClassB	sMethod()	rMethod()	Unexpected arraysize	Approved!	

SYSTEM TESTING

FUNCTIONAL REQUIREMENTS

FR1: The application should be able to save the information relevant to a specific Gmail account.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	We decided against implementing this functionality due to limited time.

FR2: The application should be able to load the previous messages from saved contact when opening a conversation window.	
Executor:	Development group
Date:	23.04.2010
Time used:	5 minutes.
Evaluation:	Works when the database is connected, but not at the moment.

FR3: The application should be able block contacts.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	We decided against implementing this functionality due to limited time.

FR4: While the user is in a conversation, the application should listen for any new messages from the contact.	
Executor:	Development group
Date:	23.04.2010
Time used:	4 minutes
Evaluation:	Works! The application runs a thread dedicated to listening for incomming messages. While the application is outside a conversation with a given chat partner, the received messages is still saved in the database and gets displayed whenever the user chooses to open the conversation window.

FR5: The application should let you know if a contact has sent you a message.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	We decided against implementing this functionality due to limited time.

FR6: The application should run on Android 1.6	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	

FR7: The application should allow users to send files in IM conversations.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	We decided against implementing this functionality due to limited time.

FR8: The contact list should be dynamically updateable.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	We decided against implementing this functionality due to limited time. You will need to go back and open the contactlist again.

FR9: The application should be able to add and remove contacts.	
Executor:	Development group
Date:	23.04.2010
Time used:	N/A
Evaluation:	The application automaticly accepts new contacts. We haven’t implemented the functionality required to add or remove new contacts. This has to be done from mail.google.com

FR10: The user should have access to old conversations while offline.	
Executor:	Development group

Date:	23.04.2010
Time used:	N/A
Evaluation:	You won’t be able to access the application without internet.

FR11: The user should only be available to send message containing string.	
Executor:	Development group
Date:	23.04.2010
Time used:	1 minute
Evaluation:	Our outgoing communication is formatted as strings.

FR12: The application should be able to connect to talk.google.com	
Executor:	Development group
Date:	23.04.2010
Time used:	1 minute
Evaluation:	The XMPP connection works and we are able to connect against talk.google.com

FR13: Each line of the conversation should have a timestamp with the corresponding date, month and time. Formatted as “DD-MM ‘at’ HH:MM”.	
Executor:	Development group
Date:	23.04.2010
Time used:	1 minute
Evaluation:	The conversation is formatted correctly.

FR14: Conversations should be saved in a database containing information about both chat partners and who sent what message.	
Executor:	Development group
Date:	23.04.2010
Time used:	4 minutes
Evaluation:	The database is ready for use, but not connected at the moment, due to changes in the code and limited time.

FR15: When displaying the contact list, each contact's email and status should be visible.	
Executor:	Development group
Date:	23.04.2010
Time used:	3 minutes
Evaluation:	Works! The email to contact is visible, and an icon beside the email represent if the contact is online or offline.

QUALITY REQUIREMENTS

T1: Test the ability to write to the database by having the controller interact with the relevant model	
Executor:	Development group
Date:	23.04.2010
Stimuli:	
Expected response:	
Observed response:	
Evaluation:	

Comments:

T2: Test the Internet connection	
Executor:	Development group
Date:	23.04.2010
Stimuli:	Press test connection button.
Expected response:	Get a response on the internet status, ok or failed.
Observed response:	Works, when we don’t have internet access it fails and its success when we have a connection.
Evaluation:	Test if the user have either a 3G or Wi-Fi connection

Comments: Note: In the emulator it tests by default 3G first.

M1: Add functionality for taking a new photo with the mobile camera and adding it as a personal picture associated with the account	
Executor:	Development group
Date:	
Stimuli:	
Expected response:	
Observed response:	
Evaluation:	

Comments:

U1: Attempts to start a conversation with an online friend

Executor:	Development group
Date:	23.04.2010
Stimuli:	Log in a Gmail account, and press on an online contact. Press send message.
Expected response:	Chat window appear contact receive message.
Observed response:	Chat window appeared with sent message containing a timestamp.
Evaluation:	Works great

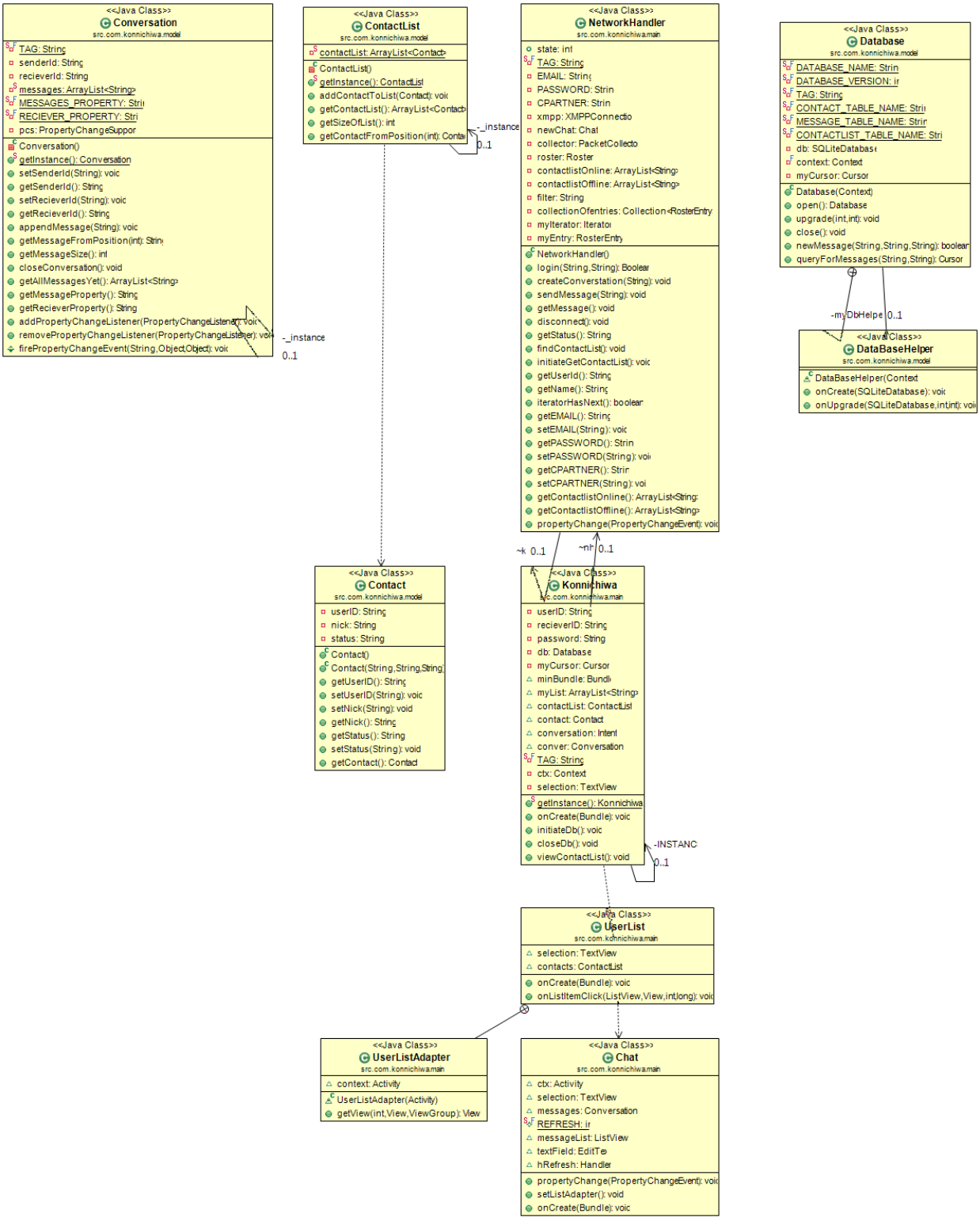
Comments:

A1: Opens the program while offline	
Executor:	Development group
Date:	23.04.2010
Stimuli:	Run the application.
Expected response:	Logon window appear.
Observed response:	Logon window appeared.
Evaluation:	Works.

Comments:

RELATIONSHIP WITH ARCHITECTURE

FINAL ARCHITECTURE



COMMENTS

The 'Controller' part of our architecture consists of the classes Chat, Konnichiwa, NetworkHandler
and UserList.

The 'Model' part of our architecture consists of the classes Contact, ContactList,

Conversation and Database.

Our final architecture differs from the architecture presented at the previous iteration, but this is given due to the fact that our understanding of the architecture patterns improved.

PROBLEMS,ISSUES AND POINTS LEARNED

We started late with the implementation, this was because of the school trip to japan, and miscommunication. Also there were problems with our svn, this resulted to time wasted. Another thing was we started implementing Smack, but we didn't get it to work. So we switched to aSmack and things went smoother.

So at the end of this iteration we had to sacrifice some functionality to make the ends meet. A lesson we definitely learned is that getting comfortable with new technologies (SVN, Android) and design methods (MVC and other Architectural Patterns) takes more time than you think.

We also learned that it's important to know where our focus should lie throughout the whole implementation phase. We had to scrap a lot of code due to the fact that we derailed from some of the architectural patterns we choose to use. In this first version of our application the data-flow didn't follow the MVC pattern. But in the final revision we implemented it correctly (at least we thought) together with other patterns (Singelton first and for most). Another thing we realized is that Singelton makes sharing data with other classes a lot easier.

One of our security scenarios was have penetration test, we didn't have enough time to implement excessive security or test for it. In the midst of development we had to make changes on how the application should behave and work, and this has made many of our scenarios meaningless. Though the application has more or less the same functionality.